Certificación PCAP

Lección 2: Operadores, estructuras de control y funciones

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Lección 2: Tipos de datos, operadores, estructuras de control y funciones

1. Objetivos

El objetivo que perseguimos en esta segunda lección es el comienzo del repaso del temario que cubre el examen de la certificación PCAP. Hasta ahora solo hemos hablado de aspectos generales de la misma, pero a partir de ahora haremos más énfasis en la parte teórica, sobre todo los puntos que tienen una mayor probabilidad de aparecer en el examen final.

Si hemos sido un poco curiosos, y nos ha dado por "ojear" el material de los cursos **Python Essentials 1 (Basics, v.2.0)** y/o su hermano mayor el **Python Essentials 2 (Intermediate, v.2.0)**, posiblemente hayamos podido comprobar que tanto en OpenEDG como en Cisco Networking Academy se nos hace una estimación en horas del tiempo que nos llevaría estudiar todos los apartados del curso.

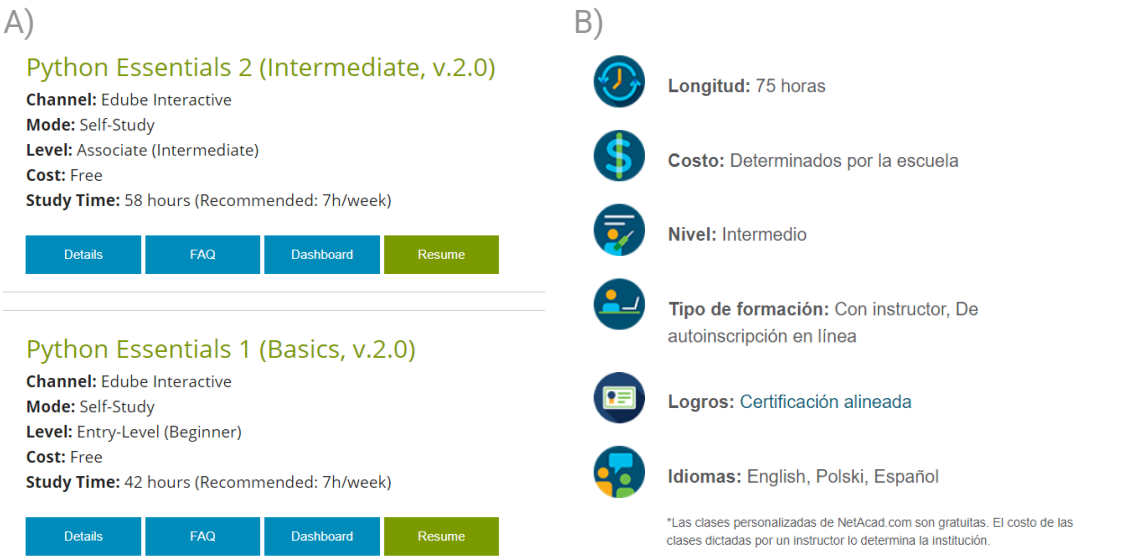

Figura 1. Resumen información sobre los cursos orientados al PCAP de A) OpenEDG y B) Cisco Networking Academy

Podemos oberservar como la estimación de horas no coincide en ambos casos, posiblemente por las diferencias en la manera de plantear el curso, pero de lo que sí podemos estar seguros es de que para poder estudiar minuciosamente todos los contenidos de los cursos y por ende, todo el contenido cubierto por el examen, necesitaríamos aproximadamente entre unas 75 y 100 horas.

Como es lógico, y puesto que este no es el planteamiento de la presente asignatura, nuestra objetivo durante estas clases no será el estudio minucioso de todos los detalles del temario, sino que nos centraremos en los que podrían considerarse los aspectos más importatnes y sobre todo, aquellos que aparecen con más frecuencia en el examen.

Dicho lo cual, durante esta lección tendremos como objetivo repasar los aspectos fundamentales de los literales, las variables, los operadores y expresiones, las estructuras de control y las funciones, las cuales considero, desde un punto de vista personal, son muy recurrentes en las preguntas del PCAP.

2. Literales de Python

Vamos a comenzar la lección repasando el término **literal**. Un literal se refiere a datos cuyos valores están determinados por el mismo literal, dicho con otras palabras, los literales son una notación para representar un valor fijo en el código fuente. Python tiene varios tipos de literales, por ejemplo, un literal puede ser un número o una cadena. Ahora que ya sabemos qué es un literal vamos a repasar los diferentes tipos de literales que ofrece Python.

2.1. Los números enteros

Los **números enteros** son aquellos que no tienen decimales, tanto positivos como negativos (además del cero). En Python se puede representar mediante el tipo *int* (de integer, **entero**) o el tipo *long* (**largo**). La única diferencia es que el tipo long permite almacenar números más grandes. Para no malgastar memoria es aconsejable no utilizar el tipo long a menos que sea necesario.

Para poder conocer el tipo de objeto que guarda una variable, en Python existe el método **type()** que te devuelve dicha información.

```
1. >>> entero = 25
2. >>> type(entero)
3. <type 'int'>
```

Para forzar que Python guarde un valor como long se puede indicar con una L al final:

```
1. >>> entero = 25L
2. >>> type(entero)
3. <type 'long'>
```

Indicar que desde la versión de Python 3.6 se puede separar los dígitos con un guion bajo:

```
1. >>> entero = 1_000_000
2. >>> print(entero)
3. 1000000
```

Por defecto, Python almacena los números enteros en representación decimal, pero también permite utilizar los números en su representación octal y hexadecimal.

Si un número entero está precedido por un código 0o o 0O (cero-o) dicho número será tratado como octal lo que significará que el número debe contener sólo dígitos en el rango del [0..7].

```
1. >>> entero_octal = 0o342
2. >>> print(entero_octal)
3. 226
```

Como podemos ver la función print realiza automáticamente la conversión a representación decimal.

Por otra parte, si un número está precedido por el código 0x o 0X (cero-x) dicho número será tratado como hexadecimal.

```
1. >>> entero_hexadecimal = 0x3A
2. >>> print(entero_hexadecimal)
3. 58
```

2.2. Los números flotantes

Una vez vistos los números enteros, vamos a pasar a conocer otro tipo numérico que nos ofrece Python, los números **flotantes**. Estos literales permiten almacenar valores numéricos que tienen una **parte decimal no vacía**.

```
1. >>> flotante = 3.5
2. >>> type(flotante)
3. <type 'float'>
```

Cuando la parte entera o la parte flotante son 0, Python también permite representarlos omitiendo el 0.

```
1. >>> flotante = 4.
2. >>> type(flotante)
3. <type 'float'>
4. >>> flotante2 = .4
5. >>> type(flotante2)
6. <type 'float'>
```

Otra forma de representación de valores flotantes que permite Python es la notación científica (base x $10^{exponente}$), donde el exponente se representa después del código *E* o *e* (baseEexponente).

```
1. >>> flotante = 4E2
2. >>> print(flotante)
3. 400.0
4. >>> flotante2 = 4.2E2
5. >>> print(flotante2)
6. 420.0
```

2.3. Las cadenas de caracteres

Las cadenas de caracteres, son secuencias inmutables que contienen caracteres encerrados entre comillas o apóstrofes. Indicar que si una cadena se delimita con una comilla, se debe cerrar con una comilla y si se delimita con un apóstrofe, se debe cerrar con un apóstrofe.

```
1. >>> cadena1 = "Mi primera cadena"
2. >>> type(cadena1)
3. <type 'str'>
4. >>> cadena2 = 'Mi segunda cadena'
5. >>> type(cadena2)
6. <type 'str'>
```

2.4. Los valores Booleanos

Otro tipo de literal que tiene Python serían los valores booleanos que se emplean para representar la **veracidad**. El tipo booleano sólo puede tener dos valores: True (verdadero, 1) y False (falso, 0).

```
1. >>> booleano1 = True
2. >>> type(booleano1)
3. <type 'bool'>
4. >>> booleano2 = False
5. >>> type(booleano2)
6. <type 'bool'>
```

2.5. El literal None

El último tipo de literal que vamos a ver es un literal especial que se utiliza para representar la **ausencia de valor**.

```
1. >>> ausencia = None
2. >>> type(ausencia)
3. <type 'NoneType'>
```

3. Operadores y expresiones

En esta sección repasaremos el funcionamiento, características principales y pequeñas sutilezas de los operadores y las expresiones que podemos formar con ellos.

3.1. Operadores y expresiones

¿Por qué hablar de operadores y expresiones si son algo tan "básico"? Pues bien, resulta que este tema es bastante recurrente en las preguntas del examen, ya que no solo se pueden preguntar directamente, es decir, no solo pueden entrar preguntas del estilo *"¿Cuál es el resultado de la siguiente expresión?"* sino que también pueden aparecer de manera indirecta en cualquier otro tipo de preguntas relacionadas con variables, funciones, bucles, tratamiento de datos, etc.

Por esta razón es por lo que creemos conveniente repasar los principales operadores existentes en Python junto con la "**jerarquía de prioridades**", ya que, si tenemos suficientemente clara esta jerarquía y el funcionamiento de los operadores, será bastante difícil que nos equivoquemos en cualquiera de las preguntas en las que estos entren en juego.

Aunque sea muy básico, lo primero que debemos hacer es definir lo **operador y expresiones**. Se entiende, pues, por operadores a un conjunto de *símbolos especiales* que, como su nombre indica, *operan* sobre los llamados **operandos**, que al fin y al cabo no son nada más que valores. Combinados, operadores y operandos forman **expresiones** y estas expresiones tienen como resultado un valor que puede ser calculado.

Como se podrá intuir, el número de operadores existentes es bastante extenso, pero nos centraremos principalmente en los más "usuales" y como siempre, recurrentes en el examen. De tal manera que repasaremos los operadores aritméticos, los lógicos, los de comparación, los de comparación bit a bit y finalmente, también los operadores de asignación.

3.2. Operadores aritméticos

Como su nombre indica, estos operadores están relacionados con las operaciones matemáticas fundamentales. Estas operaciones son: (+, -, *, /, //, %, **), suma, resta, multiplicación, división, división entera, módulo y exponenciación, respectivamente.

Hablaremos ahora de las reglas que rigen el comportamiento de cada uno de estos operadores.

Suma (a + b), resta (a - b), multiplicación (a * b), división (a / b)
El comportamiento de estos operadores es el mismo al que ya conocemos del ámbito de las matemáticas, no hay misterio alguno. Aun asi, cabe destacar que cuando realizamos estas operaciones debemos tener en cuenta cuál es el tipo de datos de cada uno de los operandos que entran en juego estas expresiones. Es decir, los números que se utilicen en estas expresiones bien pueden ser todos enteros o bien todos flotantes, o el caso que nos encontraremos en las preguntas del examen, combinaciones de ambas. En estos casos es importante conocer las reglas que sigue Python para determinar no solo el valor sino también el tipo de dato del resultado de la expresión. Las reglas que rigen el comportamiento de estas operaciones son bastante sencillas, pero con un pequeño detalle que merece la pena mencionar con respecto al comportamiento del operador división.

- En el caso de los operadores (+, -, *,**), si uno de los dos operandos es un flotante, el resultado también lo será.
- Por el contrario, en el caso del operador división (/) el resultado **siempre** será un flotante, aunque ambos operandos sean enteros.

```
1. >>> print(2+3)
2. 5
3. >>> print(5-2)
4. 3
5. >>> print(3*2)
6. 6
7. >>> print(12/4)
8. 3.0
```

Operador división entera (//)

Seguramente, al leer que el resultado del operador division siempre da como resultado un flotante, podrá haber surgido la pregunta de *¿y qué pasa si me interesa que el resultado de la división sea un entero?* Es aquí donde entra en juego este operador y de ahí su nombre. Este operador devuelve la división redondeada al entero **inferior** más cercano al resultado de división normal y en el caso de que ambos operandos sean enteros, el resultado también lo será. Lo mejor para poder entender su funcionamiento es un ejemplo.

```
1. >>> print(3//2)
2. 1
3. >>> print(3//2.0)
4. 1.0
```

Como se puede observar en el primer caso se cumple que ambos operandos son enteros, y el resultado de 3/2 es 1.5. Siguiendo la regla de que se redondea este resultado al entero inferior mas cercano, llegamos al resultado final.

El segundo ejemplo es similar, con la excepción de que el segundo operando no es un entero, sino un flotante, aplicando las mismas reglas, llegamos a que el resultado es el flotante 1.0.

Recalcamos que también es clave tener siempre presente que el redondeo es hacia el entero inferior y se aplica, por supuesto, la misma regla cuando los numeros son negativos. Otro ejemplo donde podemos ver lo que no es correcto vs lo que sí lo es.

```
1. >>> # ~~-3 // 2 = -1~~ ¡Ojo, es el entero inferior!
2. print(-3//2)
3. -2
```

Es importante estar atento a estos pequeños detalles, ya que puede ser muy fácil que se nos pase por alto por estar muy acostumbrados a tratar con divisiones en el día a día y fallar en las preguntas en las que entren expresiones similares a estas, posiblemente un poco más complejas pero con el mismo fundamento teórico.

Operador exponenciación (**)

Este operador es el homólogo de la **potencia** en matemáticas. Su trabajo consiste en elevar el primer operando a la potencia del segundo operando. Aquí el ejemplo.

```
>>> print(2**3)
8
>>> print(3**2.0)
9.0
```

```
>>> print(type(2**2))
<class 'int'>
>>> print(type(2**2.0))
<class 'float'>
```

Como se puede observar en este ejemplo anterior, en el caso de la exponenciación, se sigue cumpliendo la regla de que si uno de los operandos es un flotante, el resultado también lo será mientras que si los dos son enteros, el resultado será entero.

Operador residuo o módulo (%)

Este operador se aleja un poco de los tradicionales operadores matemáticos y no tiene un equivalente como tal. Aun asi, su comportamiento es bastante sencillo de entender, ya que la operación **a%b** se encarga de devolvernos el resto de la división de **a** entre **b**. Este operador puede resultarnos de mucha utilidad a la hora de realizar diferentes tareas, y además suele dar mucho juego para formular expresiones simples que debamos resolver en el examen del PCAP. Como siempre, un ejemplo para verlo de manera práctica.

```
>>> print(5%4)
1
>>> print(7%5)
2
```

3.3. Operadores de comparación

Otro de los tipos de operadores que son muy recurrentes, y ya no solo en los ejercicios del examen PCAP, sino en cualquier desarrollo que queramos construir con Python son los operadores de comparación, que, como su nombre indica se encargan de comparar dos operandos y devolver un valor en función de cuál sea esta comparación. Estos operadores suelen ser fundamentales para el control de las ejecuciones de condicionales y bucles y por ello los repasaremos brevemente.

Los operadores de comparación más utilizados son los siguientes

==	Igual
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
!=	Distinto que

Tabla 1. Operadores de comparación más utilizados o comunes

Este conjunto de operadores también resulta bastante fácil de "dominar" ya que se explican prácticamente con solo mencionar su nombre. Fundamentalmente, estos operadores comparan dos operandos y devuelven el **valor booleano** "True" si la comparación es verdad y en caso contrario el booleano "False". A continuación, algunos ejemplos.

```
1. >>> print(1 == 1)
2. True
3. >>> print(1 < 2)
4. True
5. >>> print(4 > 3)
6. True
7. >>> print(1 >= 0)
8. True
9. >>> print(7 <= 7)
10.True
11.>>> print(2 != 2)
12.False
13.>>> print(type(2!=2))
14.<class 'bool'>
```

Como se puede observar en la línea 14 del ejemplo, el resultado que devuelven estos operadores es de tipo **bool** (a menos que uno de los operandos sea el resultado de una operación que contenga algún error como una división entre 0, en cuyo caso la comparación arrojaría un objeto de error). Es muy importante que tengamos claro qué **tipo** de dato devuelven como resultado no solo estos, sino todos los operadores, ya que se hace mucha insistencia en ello en el examen.

Python también permite comparar valores que no son del mismo tipo. Veamos algunos ejemplos:

```
1.    >> print(2 == 2.0)
2.    True
3.    >>> print(2 == "2")
4.    False
```

En el primer ejemplo, lo que ocurre internamente es que el valor flotante se convierte en entero y por ello el resultado es verdadero. En cambio, en el segundo caso, Python no convierte las cadenas y por ello el resultado es falso.

3.4. Operadores lógicos

Pasemos ahora a un nuevo tipo de operadores, los llamados operadores lógicos, son los siguientes.

and	Operador lógico de conjunción
or	Operador lógico de disyunción
not	Operador lógico de negación

Tabla 2. Operadores lógicos

La particularidad de este tipo de operadores es que trabaja con valores booleanos, es decir "True" o "False" y en función de cuál sea el operador utilizado devuelve, igualmente, un valor de tipo bool.

De manera resumida podemos decir que:

- El operador **and** solo devuelve True en caso de que ambos operandos sean True, en cualquier otro caso devuelve False.
- El operador **or** devuelve True en el caso de que alguno de los operandos sea True.
- El operador **not** es un operador unario (solo necesita de un operando para "funcionar") y se encarga principalmente de transformar True en False y viceversa.

Como siempre, algunos ejemplos serán más esclarecedores.

A	B	A and B
True/1	True/1	True/1
True/1	False/0	False/0
False/0	True/1	False/0
False/0	False/0	False/0

A	B	A or B
True/1	True/1	True/1
True/1	False/0	True/1
False/0	True/1	True/1
False/0	False/0	False/0

A	Not A
True/1	False/0
False/0	True/1

Tabla 3. Tablas resumen del comportamiento de los operadores lógicos

En este caso, puesto que los operandos solo pueden tomar dos valores, es fácil registrar cuáles serán todas las posibilidades que nos podremos encontrar a la hora de trabajar con ellos.

3.5. Operadores de comparación bit a bit (bitwise)

Los operadores de comparación bit a bit, son un tipo de operador un tanto diferentes a los que venimos viendo hasta ahora, ya que permiten realizar comparaciones a nivel de bits individuales de los operandos implicados en la operación de comparación. Los operadores de comparación bit a bit son los siguientes.

&	Conjunción
|	Disyunción
^	Intercalación, xor o disyunción exclusiva
~	Negación

Tabla 4. Operadores de comparación bit a bit

Como podemos observar, tenemos los operadores análogos a los operadores lógicos y un nuevo operador añadido llamado intercalación, "xor" o también conocido como disyunción exclusiva, la cual solo devuelve el valor booleano True en el caso de que **uno y solo uno** de los operandos sea 1 o True. A continuación, resumimos brevemente el comportamiento de cada uno de estos operadores.

A	B	A & b	A | B	A ^B
1/True	1/True	1/True	1/True	0/False
1/True	0/False	0/False	1/True	1/True
0/False	1/True	0/False	1/True	1/True
0/False	0/False	0/False	0/False	0/False

Tabla 5. Resumen del comportamiento de los operadores de comparación bit a bit

Nuevamente, merece la pena destacar ciertas sutilezas que son muy susceptibles de aparecer en alguna de las preguntas del examen. Como se puede observar en las tablas que tienen que ver con operaciones con valores booleanos, para Python 1 es equivalente a True y 0 a False (al menos para este tipo de operaciones). A continuación, algunos ejemplos donde vemos cómo se puede combinar el uso de 1, 0, True y False obteniendo el resultado esperado en cada caso.

```
1. >>> print(1 == True)
2. True
3. >>> print(0 == False)
4. True
5. >>> print(1 & True)
6. 1
7. >>> print(0 | True)
8. 1
9. >>> print(0 or False)
10. False
```

```
>>> print(1 and True)
True
>>>
```

Estos operadores no solo pueden utilizarse con bits individuales, sino que pueden aplicarse también entre valores enteros.

La manera en la que Python los tratará será primero convertirlos a binario y posteriormente realizar las comparaciones de cada uno de los bits de los enteros que hayamos utilizado.

Muy importante tener en cuenta que estas operaciones devuelven un error en el caso de que los operandos sean flotantes.

```
>>> print(4 ^ 2) # 4 == 100, 2 == 10 en binario
6
>>> print(4 ^ 2.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand
type(s) for ^: 'int' and 'float'
```

3.6. Operadores de asignación

Finalmente, veremos otro tipo de operadores, los llamados operadores de asignación, cuyo nombre, nuevamente, nos proporciona bastante información en relación a la función que desempeñan este tipo de operadores. Estos operadores se encargan de asignar valores a las variables. En la siguiente tabla podemos una lista con bastantes de ellos, junto con una breve explicación del resultado que devuelve cada uno de ellos.

Símbolo	Funcionalidad
=	a = 5. El valor 5 es asignado a la variable a
+=	a += 5 es equivalente a = a + 5
-=	a -= 5 es equivalente a = a - 5
*=	a *= 3 es equivalente a = a * 3

/=	a /= 3 es equivalente a = a / 3
%=	a %= 3 es equivalente a = a % 3
****=**	a **= 3 es equivalente a = a ** 3
//=	a //= 3 es equivalente a = a // 3
&=	a &= 3 es equivalente a = a & 3
|=	a |= 3 es equivalente a = a | 3
^=	a ^= 3 es equivalente a a = a ^ 3

Tabla 6. Símbolo y funcionalidad de los operadores de asignación

Como podemos ver, todos los operadores de asignación parten del "operador de asignación simple" (si quisiéramos llamarlo de esta manera) que es el operador "**=**", cuyo funcionamiento es bastante sencillo, asigna un valor a una variable.

Este operador de asignación simple, puede combinarse con alguno de los operadores vistos en apartados anteriores (generalmente con los aritméticos, aunque no son los únicos), dando lugar a operadores de asignación con un comportamiento descrito en cada uno de los casos de la Tabla. Una de las finalidades que tienen estos operadores es la reducción de la cantidad de código que sería necesario escribir en el caso de que dichos operadores no existieran.

Puesto que estas expresiones en las que se modifica el valor de una variable suelen estar bastante presentes dentro de las estructuras de control de ejecución *i.e.* bucles y condicionales, también será muy probable que no pocas preguntas del examen contengan alguno de estos operadores.

3.7. Jerarquía de prioridades de operadores

Hasta aquí hemos estudiado cómo se comporta cada operador, cuál es su función y si tiene alguna particularidad que debamos tener en mente al trabajar con ellos, pero en todo momento hemos puesto ejemplos de operadores en situaciones aisladas, es decir, donde interviene solo un operador. Por más que esta situación sea totalmente usual en el día a día de un desarrollador, debemos recordar en todo momento que nuestro objetivo principal en esta asignatura es prepararnos para poder superar un examen

de certificación y, como es lógico, en dicho examen, no será tan usual, por no decir que será casi imposible, encontrar expresiones tan simples como estas. La situación real del examen será encontrarnos con expresiones que contengan varios operadores de diferentes tipos para que hallemos el resultado final. Es por ello que, como mencionamos al comienzo de esta sección, es muy importante conocer cuál es la **jerarquía de prioridades** entre los distintos operadores, ya que el resultado final se verá claramente afectado según el orden en el que realicemos las operaciones.

La siguiente tabla nos ayudará a resumir cual es esta jerarquía. Es altamente recomendable aprenderla y/o memorizarla, no nos tomará mucho tiempo y retribuirá en mucho beneficio en las preguntas del examen. Encontraremos los operadores situados de mayor (primeras filas) a menor (últimas filas) prioridad.

operadores	descripción
**	Exponenciación (prioridad más alta)
~, +, -	Negación bit a bit, más y menos **unario**
*, /, //, %	Multiplicación, división, división entera y módulo
+, -	Suma Resta
>> , <<	Desplazamiento de bits
&	Conjunción bit a bit
^ |	Disyuntiva y xor bit a bit
<=, <=, >>	Comparación
==, ! =	Operador de igualdad
=, +=, -=, /=, //=, %=, * =, **=	Operadores de asignación

Tabla 7. Jerarquía de prioridad de los operadores en Python

Teniendo presente la prioridad que tiene cada uno de estos operadores podemos conocer el correcto orden de actuación de los mismos, pero, aun así, tendríamos un inconveniente más que resolver. ¿Qué ocurre con los operadores que comparten prioridad, como es el caso, por ejemplo, de la multiplicación, división, división entera y módulo?

Introducimos, por lo tanto, una nueva propiedad de los operadores, el **enlace**. Esta propiedad se define como el orden en el que los operadores con misma prioridad deben actuar cuando se encuentran presentes en la misma expresión.

Este orden es muy simple de recordad, ya que salvo el operador exponenciación, el resto de operadores se ejecutan tal y como lo harían en una expresión matemática, de derecha a izquierda. Podemos comprobarlo en el siguiente ejemplo.

```
1. >>> print(3 * 4 / 6)
2. 2.0
3. >>> print(4 / 6 * 2)
4. 1.3333333333333333
```

Como vemos en el ejemplo, el resultado puede verse muy alterado si no tenemos claro el orden de aplicación de los operadores. Los operadores que tienen este tipo de enlace *i.e.* de izquierda a derecha, se dice que tienen enlace hacia la izquierda.

Ahora veremos un ejemplo con la operación de exponenciación

```
1. >>> print(2 ** 3 ** 2)
2. 512
3. >>> print(2 ** 2 ** 3)
4. 256
```

Como vemos, $2 ** 3 ** 2 \neq 64$ como posiblemente hubiéramos esperado, sino que en el caso de que tengamos dos operadores de exponenciación situados juntos, la prioridad de ejecución será contraria a la del resto de operadores. Los operadores que tienen este tipo de enlace u orden *i.e.* de derecha a izquierda, se dice que tienen enlace hacia la derecha. No importa tanto el nombre, sino que tengamos muy en cuenta esta nueva sutileza que podemos encontrar en algunas de las expresiones que aparezcan en el examen.

4. VARIABLES EN PYTHON

Aunque en algunos ejemplos de los apartados anteriores ya se han utilizado variables, ahora es momento de presentar este concepto. Una **variable** es una ubicación nombrada reservada para almacenar valores en la memoria.

Una variable está formada por un nombre único, **identificador**, y un valor. Para nombrar una variable, se deben seguir las siguientes reglas:

- El nombre de la variable sólo puede estar compuesto por MAYÚSCULAS, minúsculas, dígitos y el carácter '_' (guión bajo).
- El nombre de la variable debe comenzar con una letra o el carácter '_', pero no por dígitos.
- Las mayúsculas y minúsculas se tratan de forma distinta.
- No hay restricciones respecto a la longitud.
- Se pueden utilizar caracteres específicos que utilizan otros alfabetos.
- El nombre de las variables no puede ser una de las palabras reservadas de Python.

Se acaba de comentar que el nombre de una variable no puede ser una de las palabras reservadas de Python. Pero cuáles son las palabras reservadas de Python, éstas son las siguientes:

'False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield'

Es importante saber que en Python una variable se crea cuando se le asigna un valor. A diferencia de otros lenguajes de programación, no es necesario declararla. En el siguiente ejemplo podemos ver cómo se crearía la variable 'mi_variable' y además podemos ver que si se intenta utilizar una variable que no existe, se devuelve un Error.

```
1. >>> mi_variable = 3
2. >>> print(mi_variable)
3. 3
4. >>> print(Mi_variable)
5. NameError: name 'Mi_variable' is not defined
```

5. Estructuras de control (condicionales y bucles)

En este apartado repasaremos el funcionamiento de las principales estructuras de control de ejecución, es decir, los condicionales y los bucles. Nuevamente, estas estructuras son muy habituales en cualquier pieza de código, prácticamente indispensables para cualquier desarrollo, con lo cual conocer su funcionamiento y principales rarezas será de mucha ayuda tanto para afrontar el examen como para nuestro futuro desempeño profesional.

Como bien sabremos por estar familiarizadas con ellas, las estructuras de control, como su nombre indica, nos permiten controlar o modificar el flujo de ejecución de las instrucciones de un programa. Son fundamentales para poder dotar de "interactividad" a los mismos y nos permiten, entre otras cosas, tomar decisiones en función del resultado de determinada expresión que nos interese evaluar (de ahí el estudio de los operadores del apartado anterior) o bien, controlar en número de veces que han de ejecutarse determinadas instrucciones.

5.1. Condicionales (if)

Esta estructura se parece mucho al "si" (sin tilde) condicional de nuestra lengua, ya que expresa una condición que ha de cumplirse para el desarrollo o no de una actividad, de ahí su nombre. En Python, esta estructura es muy similar, con sus propias particularidades, claro, pero muy similar, al fin y al cabo.

La palabra reservada para los condicionales es el "**if**" y la sintaxis es la siguiente

```
1. if condicion1:
2.         instruccion1
3. elif condicion2:
4.         instruccion2
5. else:
6.         instruccion3
7.
```

Puntos a tener en cuenta a la hora trabajar con condicionales:

- **Indentación y sintaxis**. Python es muy estricto con la sangría en las estructuras de control, de ahí que sea tan amigable a la vista y rápido de entender. Además, es imprescindible la presencia de los dos puntos ":" para denotar el final de la expresión de la condición.
- La condición que se desea evaluar puede ser, y lo es en la mayoría de los casos, una expresión lógica con un resultado de tipo bool, True o False. Aun así, no solo es posibe utilizar este tipo de expresiones como veremos en el siguiente ejemplo, sino que también son válidas las expresiones matemáticas de resultado numérico, string o practicamente cualquier variable.

```
>>> if 2*3:
...     print("Una expresión matemática es válida")
...
Una expresión matemática es válida
>>> if "esto es un string":
...     print("Un string también es válido")
...
Un string también es válido
>>> variable = ["elemento1",2]
>>> if variable:
...     print("Un array también puede utilizarse")
...
Un array también puede utilizarse
>>> if []:
...     print("Un array vacío")
```

En estos caso en los que la expresión a evaluar no arroja un resultado de tipo bool, o directamente no se trata de una expresión sino de una variable, la manera en la que actua el "if" es **comprobar que la variable no esté vacía**, y si es así, Python lo interpreta como que la condición se ha cumplido, con lo cual ejecutará las instrucciones presentes dentro del condicional.

Este tipo de condiciones pueden resultar verdaderamente útiles para comprobar la existencia de una variable, o saber si esta tiene algún tipo de contenido y actuar en consecuencia. Si bien este comportamiento no es exclusivo de python, no todos los lenguajes lo toleran, como podría ser el caso de java, donde este tipo de "condiciones" arrojarían inmediatamente un error. Por lo cual, merece la pena tener en cuenta esta particularidad o comportamiento.

Sentencia "elif". Python no dispone de sentencia "switch", como podria ser el caso de php o javascript, es aquí donde entra en juego la sentencia "elif". Traducido a nuestro lenguaje cotidiano podría ser algo similar a "*si esta condición no se cumple, ¿se cumple esta otra?*". La sentencia "elif" solo entra en acción cuando la condición anterior no se ha visto cumplida, por lo cual, no tiene sentido que aparezca sola, sin la presencia de un "if" que le preceda, situación ante la cual arrojará un error.

```
>>> elif 1>0:
  File "<stdin>", line 1
    elif 1>0:
       ^
SyntaxError: invalid syntax
>>> if 1<0:
...   pass
... elif 1>0:
...   print("1 es mayor que 0")
...
1 es mayor que 0
>>>
```

Sentencia "else". Si ninguna de las condiciones anteriores es verdadera se ejecuta el codigo dentro del bloque else. Traducido a palabras un poco más usuales sería "si ninguna de las condiciones anteriores se cumple, finalmente haz esto". Al igual que la sentencia "elif", no tiene sentido que esta aparezca sin la precedencia de al menos un "if".

Si alguna de las condiciones es verdadera, el resto no se ejecutara. Es importante tener que esta es la principal diferencia entre un bloque de sentencias if-elif-else y otro de if-if.

5.2. Bucles "while" y "for"

El bucle "while" sirve principalmente para ejecutar determinadas instrucciones **mientras** la condición evaluada sea verdadera. Su sintaxis es la siguiente.

```
while condicion_a_evaluar:
        Instruccion_1
```

La condición que se debe verificar para poder ejecutar este bucle puede ser, al igual que en el condicional, una variable. Si esta variable tiene algún contenido, se ejecutará el cuerpo del bucle, en caso contrario, no. Conviene tener cuidado con este tipo de bucles ya que en caso de que la condición sea True siempre, entraremos en un bucle infinito, lo cual puede causar problemas en ciertas circunstancias.

```
>>> a = 3
>>> while 2*a:
...  print(a)
...  a -= 1
...
3
2
1
>>>
```

Como vemos en el ejemplo previo, mientras que 2*a es distinto de cero, el "while" continúa ejecutándose, en el momento que 2*a == 0, se detiene.

Por otro lado, también tenemos el bucle "for". Este tipo de bucles se encargan de hacer una ejecución repetitiva pero controlada con un número definido de ciclos. Su sintaxis es la siguiente.

```
for i in range (4):
    Instrucciones
```

Donde "i" representa la variable de control que, valga la redundancia, controlará el número de veces que se ejecutará el ciclo "for". El ejemplo anterior sería equivalente al siguiente ejemplo, donde como vemos que mediante el uso del bucle for ganamos en limpieza de código y síntesis del mismo.

```
i = 0
while i < 4:
    Instrucciones
    i += 1
```

Tanto el "while" como el "for" tienen la particularidad de aceptar el uso de la cláusula "**else**". Quizás no sea muy utilizada, pero es posible y factible de preguntarse en el examen. Veamos la diferencia en el comportamiento de las combinaciones **while-else** y **for-else**.

En el caso del **while-else**, la rama "else" se ejecuta siempre una vez, independientemente de que se haya entrado o no en el cuerpo del "while". Podemos verlo en el siguiente ejemplo.

```
>>> i = 4:
>>> while i == 2:
...     print(i)
...     i -= 1
... else:
...     print("rama else", i)
...
rama else 3
>>>
```

Por otra parte, en el caso del **for-else**, esta última rama puede ver afectada su comportamiento dependiendo de si se ha entrado o no en la rama del "for". A continuación 2 ejemplos en los que podremos ver la diferencia en el valor de la variable de control.

Sí se entra en el cuerpo del "for"

```
>>> for i in range(5):
...     print(i)
... else:
...     print("else",i)
...
0
1
2
3
4
else 4
>>>
```

No se entra en el cuerpo del "for"

```
>>> i = 111
>>> for i in range(2,1):
...     print(i)
... else:
...     print("else",i)
...
else 111
>>>
```

Como podremos observar, la variable de control toma el último valor que ha tomado antes de entrar a la rama "else". Cabe destacar, una vez más, que puede que no sea muy usual ver este tipo de situaciones en desarrollos corrientes, simplemente hacemos énfasis sobre estos pequeños detalles para evitar equivocaciones en el examen.

Existe una instrucción que se puede utilizar dentro de los bucles que permite para su ejecución. Esta instrucción es "break".

```
>>> for i in range(5):
...     print(i)
...     if i == 3:
...         break
... else:
...     print("else",i)
...
0
1
2
3
```

Como se observa en el ejemplo cuando la variable de condición tiene el valor 3 se sale del bucle y no se ejecuta para el valor 4, además cuando se ha ejecutado un break no se ejecutará la condición dentro del else.

Por otra parte, en los ejemplos superiores se está utilizando una función range(), aunque las funciones se van a introducir en el siguiente apartado, vamos a ver en qué consiste esta función. La función range() genera una secuencia de números. Acepta uno o tres parámetros que deben ser de tipo entero, donde tenemos range(start, stop, step). El parámetro start es opcional y especifica el número de inicio de la secuencia (0 por defecto), el parámetro stop especifica el final de la secuencia (no está incluido) y el parámetro step es también opcional y especifica la diferencia entre los números de la secuencia (1 por defecto). Por ejemplo:

```
>>> for i in range(3):
...     print(i)
0
1
2
>>> for i in range(1,3):
...     print(i)
1
2
>>> for i in range(1,5,2):
...     print(i)
1
3
```

6. FUNCIONES

En este apartado nos centraremos en el estudio de las funciones, otro punto sumamente relevante tanto en el día a día de un desarrollador como un aspecto muy recurrente en el examen del PCAP.

Podemos definir las funciones como piezas de código que reciben unos parámetros de entrada y entregan unos resultados de salida, cumpliendo una determinada *función*. No es obligatorio que las funciones reciban estos parámetros, al igual que tampoco lo es que devuelvan resultados, pero lo más usual es que los tengan.

Más allá de esta pequeña particularidad, uno de los motivos principales (y mayores beneficios) que justifica la existencia de las funciones es la necesidad de hacer reutilizable el código desarrollado. Cuando determinadas tareas se repiten en varios lugares del desarrollo, en vez de reescribir el código cada vez que este sea necesario se recurre a la creación de funciones que pueden ser llamadas en cualquier momento ahorrando tiempo y haciendo más limpios y entendibles los desarrollos.

La sintaxis para definir una función en Python es la que sigue.

```
1. def nombre_funcion(parametro1,parametro2,…):
2.     cuerpoFuncion
3.     return resultado
4.
5. nombre_funcion(parametro1,parametro2,…)
```

> Como vemos, "def" es una palabra reservada de Python, y lo utilizaremos siempre que queramos definir una función. Además, debemos respetar, como siempre, la correcta indentación ya que todo el código que forme parte de la función deberá tener la sangría adecuada.

> El término "return" también es una palabra reservada de Python, si se utiliza sin añadir nada más hace terminar inmediatamente la ejecución de la función y regresa al punto desde el cual se invocó. Si se utiliza acompañada, se devuelve el valor de la variable que la acompaña como resultado de la función.

- Podemos ver la manera en la que se invoca una función en la línea 5 del ejemplo.
- Los nombres de las funciones deben cumplir las mismas reglas que los nombres de las variables.
- Debemos tener clara la diferencia entre **argumentos** y **parámetros**. A la hora de tratar con funciones suele ser muy recurrente utilizar estos dos términos y es importante entender las diferencias entre los mismos. Mientras que los argumentos existen fuera de la función, son variables externas a ella, los parámetros existen solo dentro de la función y una vez esta concluye su ejecución, estas "no tienen vida fuera". Los parámetros representan a los argumentos y se asigna su valor en la llamada de la función, por los valores de los argumentos.

Las funciones pueden tener como inputs la cantidad de parámetros que se desee, por lo cual es importante entender cómo se asignan estos parámetros.

En principio existen 3 maneras o métodos para pasar parámetros a las funciones: paso de parámetros posicionales, paso de argumentos con palabras clave y combinación de ambas.

Paso de parámetros posicionales

Podremos intuir por su nombre, que esta manera de pasar parámetros a las funciones tiene muy en cuenta el orden en el que se pasan estos parámetros. A continuación, un ejemplo donde podemos ver este método de asignación.

```
1. def presentación(nombre,apellido):
2.     print("Mi nombre es",nombre, apellido)
3.
4. presentación("Juan","Martínez")
5. Mi nombre es Juan Martínez
6. presentación("Martínez", "Juan")
7. Mi nombre es Martínez Juan
```

Como vemos los parámetros “nombre” y “apellido” (variables internas de la función) se asignan por orden. El primer argumento pasado se asigna a “nombre” y el segundo a “apellido”, por este motivo, el resultado de la función se ve alterado si este orden de pasada de parámetros es alterado también.

Paso de argumentos con palabras clave

En este segundo caso, el paso de parámetros ya no tiene en cuenta el orden, sino que se hace a través del nombre que cada parámetro tiene dentro de la función. A continuación, el miso ejemplo anterior, pero con este método de paso de argumentos.

```
1. def presentación(nombre,apellido):
2.     print("Mi nombre es",nombre, apellido)
3.
4. presentación(nombre = "Juan",apellido = "Martínez")
5. Mi nombre es Juan Martínez
6. presentación(apellido = "Martínez", nombre = "Juan")
7. Mi nombre es Juan Martínez
```

En este caso, no hay lugar a confusión, los argumentos se asignan a los parámetros a través del nombre que estos tienen dentro de la función.

Combinación de argumentos posicionales y palabras clave

También es posible combinar los dos métodos anteriores y pasar argumentos de manera posicional y por palabra clave. La única **regla fundamental** es que **los argumentos posicionales han de pasarse siempre primero**. A continuación, el ejemplo de cómo funciona este método de asignación.

```
1. def suma(a, b, c):
2.     print(a, "+", b, "+", c, "=", a + b + c)
3. suma(3,c = 3,b = 5)
4.
5. 3 + 5 + 3 = 11
```

> **Debemos tener en cuenta que tanto pasar un parámetro que no existe como un parámetro repetido o no pasar un parámetro necesario arrojará un error, sin importar cuál sea el método de asignación de parámetros utilizado.**

> Es posible predefinir valores de los parámetros para evitar los errores de parámetro necesario no asignado. La manera de proceder sería la siguiente.

```
1. def suma(a=0, b=0, c=0):
2.     print(a, "+", b, "+", c, "=", a + b + c)
3. suma(3,4)
4. 3 + 4 + 0 = 7
```

De cara a la certificación es muy importante entender el **alcance de los nombres** o también conocido como **scopes**. Definimos el alcance de un nombre (por ejemplo, el nombre de una variable) como la parte del código donde el nombre es reconocido correctamente.

Resumamos cuales son estos alcances, tanto desde fuera de las funciones hacia dentro del cuerpo de las mismas como en el sentido opuesto:

> Una variable que existe fuera de una función tiene alcance dentro del cuerpo de la función.

```
1. def miFuncion():
2.     print("El valor de la variable dentro de la función
   es",var)
3.
4. var = 1
5. miFuncion()
6. print("El valor de la variable var fuera de la función
   es",var)
7.
8. El valor de la variable dentro de la función es 1
9. El valor de la variable var fuera de la función es 1
```

Existe una excepción a la regla anterior. Una variable que existe fuera de una función tiene un alcance dentro del cuerpo de la función, excluyendo a aquellas que tienen el mismo nombre.

```
1. def miFuncion():
2.     var = 0
3.     print("El valor de la variable dentro de la función
   es",var)
4.
5. var = 1
6. miFuncion()
7. print("El valor de la variable var fuera de la función
   es",var)
8.
9.      El valor de la variable dentro de la función es 0
10.     El valor de la variable var fuera de la función es 1
```

Existe una manera de extender el alcance de variables. Esto se hace a través de la palabra reservada “global”. A continuación, podemos ver un ejemplo donde se puede ver cómo se modifica el valor de una variable desde dentro de la función.

```
1. def miFuncion():
2.     global var
3.     var = 2
4.     print("El valor de var dentro de la funcion es", var)
5.
6. var = 1
7. miFuncion()
8. print("El valor de la variable var fuera de la función
   es", var)
9.
10.     El valor de var dentro de la funcion es 2
11.     El valor de la variable var fuera de la función es 2
```

Existe una particularidad si tratamos con argumentos/parámetros de tipo lista debido a la naturaleza de las mismas.

Alterar el parámetro correspondiente al argumento de tipo lista no afecta a la lista.

Por el contrario, si se modifica la lista identificada por el parámetro alterando este, la lista si mostrará esa alteración. Puede parecer un poco confuso solo de palabra, en el siguiente ejemplo se aprecia la diferencia.

No se altera la lista

```
1. def miFuncion(miLista1):
2.     print(miLista1)
3.     miLista1 = [0, 1]
4.
5. miLista2 = [2, 3]
6. miFuncion(miLista2)
7. print(miLista2)
8.
9.  [2, 3]
10. [2, 3]
```

Sí se alera la lista

```
1. def miFuncion(miLista1):
2.     print(miLista1)
3.     del miLista1[0]
4.
5. miLista2 = [2, 3]
6. miFuncion(miLista2)
7. print(miLista2)
8.
9. [2, 3]
10.[3]
```

Se puede ver claramente como en la situación de la izquierda se está alterando el parámetro asignándole un nuevo valor, mientras que en la situación de la derecha se está alterando la lista que representa el parámetro, con lo cual este cambio será observado fuera de la función.

6.1. Las funciones print() e input()

Para concluir con la lección vamos a ver dos funciones muy utilizadas la función print() y la función input().

Comencemos con la función **print**(). Esta función ya la hemos estado utilizando en los ejemplos de esta lección, pero vamos a verla un poco más en profundidad. La función print() toma los argumentos que pueden ser ninguno o varios, los convierte en un formato legible para el ser humano si es necesario y **envía los datos resultantes al dispositivo de salida**. Veamos un ejemplo con varios argumentos:

```
1. nombre = "Pepe"
2. print("Buenos días", nombre, "¿qué tal está?")
3. print("Hasta luego.")
4.  Buenos días Pepe ¿qué tal está?
5.  Hasta luego.
```

Como se puede observar la función une los diferentes parámetros poniendo un espacio entre ellos, además termina la línea con un salto de línea. Si se desea modificar este comportamiento, la función print() proporciona dos parámetros de palabras clave: **sep** (separador) y **end** (final).

```
1. nombre = "Pepe"
2. print("Buenos días", nombre, "¿qué tal está?", end=". ")
3. print("Hasta luego.")
4.  Buenos días Pepe ¿qué tal está?. Hasta luego.
5. nombre = "Pepe"
6. print("Buenos días", nombre, "¿qué tal está?", sep="+",
   end=". ")
7. print("Hasta luego.")
8.  Buenos días+Pepe+¿qué tal está?. Hasta luego.
9.
```

Por otra parte, la función **input()** es **capaz de leer datos** que han sido introducidos por el usuario y pasar esos datos al programa en ejecución. Veamos un ejemplo muy sencillo:

```
1. print("Dime tu nombre:")
2. nombre = input()
3. print("Tu nombre es:", nombre)
4.
```

Como vemos en el ejemplo, el nombre que escriba el usuario se recogerá en la variable nombre. Hay que indicar que la función input() siempre devuelve una cadena de caracteres, por lo que si el usuario introduce un número, la variable que lo recoja será de tipo cadena y por lo tanto el programa si quiere trabajar con el valor como numérico deberá realizar la conversión.

```
1. print("Escribe un entero:")
2. mi_entero = int(input())
3.
```

La función input() también puede recibir un parámetro de entrada que sería el que se imprimiría en el dispositivo de salida, por lo que podríamos simplificar el ejemplo anterior:

```
1. mi_entero = int(input("Escribe un entero:"))
```